1 Introduction

1.1 Motivation

The computational needs of critical high-end computing applications are leading to the use of ever higher levels of parallelism; systems with with tens of thousands of processors are now used \[11\]; as the number of cores on a chip increases one will soon see applications that use hundreds of thousands of simultaneously executing threads.

It is widely recognized that lack of suitable programming models is a major impediment to the exploitation of these large systems. Parallel programming is a tedious, error prone, low-level activity. On the other hand, high-end systems are expensive machines that are used when computation time is a major component of time-to-solution – it is simply not possible to ignore performance in High-Performance Computing.

1.2 Current State of the Art

Large scale systems are currently programmed using libraries such as MPI \[12\]. The message passing model is generally recognized as being too low level, because of the lack of a global name space, the need to manage explicitly communication, and the use of a simple distributed control model consisting of a fixed number of independent processes. Global Array Languages such as such as Universal Parallel C (UPC) \[14\], Co-Array Fortran (CAF) \[20\] and Titanium \[10\] provide a global name space (in the form of global arrays). These languages they still provide the same distributed control model as message passing; they support simple, regular, static partitions of data to processes. Another approach is provided by data-parallel languages such as HPF \[16\], ZPL \[21\] or Chapel \[5\]. In these languages, distribution of control is largely inferred from the distribution of data. This is not a good fit to algorithms where control distribution is irregular and dynamic and does not match well the natural way users think of parallel algorithms. Languages such as OpenMP \[4\] support a control parallel model where the focus is on distribution of control, and no

\[\ast\] with contributions by Sanjay Kale and Geraud Krawezik
support is provided for distribution of data. The experience with such languages is that it is hard to achieve good performance on computers where interprocessor communication has a significant overhead; that is, in practice, on any large parallel computer. The X10 language [9] provides more control on both data and computation location – a step in the right direction. However it has a strong emphasis on asynchronous control mechanisms that lead to code that is hard to understand and hard to debug.

The design of PPL1 is indebted to many of these projects, and, in particular, to Titanium.

1.3 Design Goals

PPL1 is designed to support high performance and high productivity for high performance scientific computing. Our focus is on large scale, tightly coupled problems that arise in computational physics, chemistry, biology, etc. We pay less attention to data-intensive problems, transactional and reactive computing, distributed computing, etc. – simply because we do not believe that “one size fits all”

The following considerations are paramount in our design:

Efficiency and Scalability: A good parallel programming language for High Performance Computing should facilitate the programming of very large parallel systems with tens of thousands of processors, without unacceptable performance penalties. It should support efficiently codes with dynamic, irregular computation and communication patterns that characterize applications such as Adaptive Mesh Refinement, Particle in Cell, etc.

Semantic and Performance Transparency: The programming language should be so that it is reasonably easy for a programmer to understand the outcome of a computation specified by a parallel code, and reasonably easy to understand the expected performance of such a code. While high-level abstraction is good, abstraction should not be so that apparently insignificant changes in a program could significantly affect performance; in particular, performance should not be significantly affected by implementation strategies in the compiler or the runtime that may be different in different implementations and are not controlled by the user.

User Control: The design of efficient parallel algorithms requires algorithmic control on the partitioning of computation across the computation resources. Indeed, specifying parallel control is of the essence in the design of parallel algorithms. This does not imply that the user must always control explicitly parallelism – only that she should be capable of doing so when desired. Nor does it imply that the user must control physical resources – some level of virtualization and of allocation by the system is acceptable, indeed, desirable, if it does not hurt performance (too much).

The specification of data distribution and of communication would seem to be less essential, as it can be determined by the pattern of data accesses: In a shared memory programming model, a communication is needed whenever a value produced by one thread is consumed by another; communication is minimized by managing local memories as caches. However, long communication latencies and the lack of good latency hiding hardware may require that data be prefetched; large communication overheads may dictate that data communications by aggregated; and the lack of coherent caches or the overhead of coherence protocols may require that data location be determined
by software. The experience of OpenMP and other similar languages indicate that some user control of data distribution and communication may be required in order to achieve performance.

**Separation of Concerns and Support for Iterative Refinement:** A good parallel programming language should allow separate handling of the different concerns users face when designing parallel code. Among these concerns are: control distribution that achieves a high level of concurrency; load balancing; data distribution that achieves good locality and reduces communication; and communication scheduling that ensures that data is available when needed and avoid idle time. To the greatest possible extent, the language design should allow users to focus only on one concern at a time.

Good separation of concern will support iterative code development where a correct code can be designed with little concern for control and data distribution and iteratively tuned by improving either or both, but without having to alter the basic code structure. If a code is “embarrassingly parallel” and requires little communication, then the programmer should not have to specify communication in detail, leaving this to the compiler and run-time.

Good separation of concerns will also promote code portability. A large, widely used scientific code may be used over decades on many different platforms; it may execute on individual workstations as well as on systems with tens of thousands of processors; it is continuously updated as new algorithms are added and old algorithms are improved. Good separation of concerns enables code changes that handle different machine scales or different compute to communication ratios to affect only a small part of the code.

**Virtualization:** To the greatest possible extent, a programmer should not program directly to the physical resources of the underlying computer, but to virtual resources that are mapped to the physical machine by the run-time. This increases code portability and increases OS and run-time flexibility in allocating resources to jobs. In particular, it is essential that parallel programs not be coded to a fixed number of physical processors. The allocation of fixed size partitions on large parallel systems leads to low system utilization, because of fragmentation. More importantly, codes written to a fixed number of physical processors are hard to compose. For example, if the different codes need to progress in tandem, then the rate of progress will be dictated by the slower code. If this rate of progress changes as the computation evolves, then one will want to borrow processors from a faster code to a slower code. This is not possible if the codes are written to a fixed number of processors.

**Global Name Space and Global Control:** By *global name space* we mean that all variables are accessible by all executing threads; a value is accessed using the same global name at all threads. The use of a global name space does not prevent the use of caching, where a local copy of remote variable is created, in order to reduce communication; the language may even provide some user control on the generation and management of such copies; but the local copies do not have different names than the global copies; their use is “transparent” in the sense that one need not change the variable names used in the program. The use of a global name space is generally believed to facilitate the development of parallel code; it is essential for the separation of concerns discussed above.

By *global control* we mean that the program describes explicitly the operations performed by the
entire parallel execution – not the operations performed by one thread: For example, in developing parallel code, one may replace a sequential iterator by a parallel iterator, but one does not change the set of iterations. This makes parallel programs more legible, and supports separation of concerns.

**Composable Parallelism:** By *composable parallelism* we mean that a sequential method can be replaced by a parallel method with no changes in the environment where the method is invoked. Composable parallelism is necessary if one wishes to support an iterative development process where methods can be parallelized if found to cause performance bottlenecks.

**Backward Compatibility:** To facilitate code evolution and the adoption of a new language, it is important that this language be compatible with programs written in current popular parallel programming models. In particular, one should be able to invoke libraries written in MPI from the new language, and one should be able to invoke programs written in the new language from MPI programs; this, with acceptable overhead for the inter-language interface.

**Object Orientation:** The use of object oriented language, such as C++, Java or C#, is generally credited with increased programmer productivity. As parallel scientific codes become larger and more complex, it is important to leverage OO technologies in the HPC domain. Good OO programming style facilitates code maintenance as new algorithms and new data structures can be introduced while affecting only a small part of the code. In addition, we want to avail ourselves of the advantages of strong typing and memory safety in languages such as Java and C#, but without the performance penalties that strict reproducibility entails in Java.

### 2 Outline of PPL1

#### 2.1 Java Core

PPL1 is Java based, and inherits from Java its support for Object Orientation, strong typing, and memory safety. In addition, the similarity to Java reduces the learning curve for the very large number of programmers that are familiar with Java. However, PPL1 parts from Java when needed for performance or ease of use. In particular:

- We add new parallel control structures and do not support Java threads and the Java shared memory semantics.
- We do not insist on strict reproducibility, but allow transformations that assume that floating point operations are associative. This flexibility is essential for parallelizing reduction operations.
- We do not support Java features such as dynamic class loading that prevent static compilation.
- We do not support all the Java libraries.
- We support complex and matrix operands.
- We overload operators to apply to objects such as matrices.
• We support deep (copy) assignments for objects such as matrices.

**Missing Material**
Need to provide more detailed list of departures from Java.

Some of the changes, such as support for parallel control, are very significant and essential to our goals.

The weakening of the strict reproducibility rules is a significant departure, in principle, but is unlikely to have much practical import: Java has already weakened strict reproducibility of floating point operations, in order to accommodate transparently Intel floating point hardware; it is possible, at the expense of less transparency and more implementation complexity, to support both Java floating-point semantics and more relaxed semantics.

The modifications introducing new operands, operations and assignments can be seen as syntactic sugar that replaces awkward method invocation syntax with simpler expressions. Thus, \( a = b + c * d \), where \( a, b, c \) and \( d \) are matrices, could be rewritten as \( a \text{.assign}(b \text{.plus}(c \text{.times}(d))) \). While improved syntax is an important goal (as shown by this example), there also is an expectation that a compiler would handle such expressions directly, and avoid method invocation overheads. (The two expressions are not strictly equivalent, in fact, because of the different granularity in exception handling.)

The main departure from Java is that we do not assume the virtual machine model of Java and do not require the sandboxing provided by JVM. An initial implementation of PPL1 will use static compilation and a very different run-time. The support of a JVM-like environment for PPL1 is beyond the scope of our effort.

**Discussion**
The use of Java as a basis is not essential to our purposes; we could, instead, build PPL1 as an extension to other modern OO languages. It would be interesting to see whether PPL1 can be built as an extension to Fortress [2], providing a layer for library design and tuning.

### 2.2 Global Name Space and Communication

PPL1 provides a global names space; communication is performed via accesses and updates to shared variables, and not special syntax is used for accesses to such variables.

While the use of a shared memory model simplifies programming, it may lead to inefficient code. On many platforms it is important to perform optimizations that reduce the overhead of global data transfers – for example, aggregating small messages into fewer, larger messages; ensuring that data transfers are overlapped with computations so that computations do not have to wait for communications to complete (prefetching); caching data locally, so as to leverage temporal locality of references; and reducing the overhead of cache management (cache storage management, directory maintenance and coherence protocol). A different subset of these optimizations is relevant for different hardware architectures.
In languages that support a shared memory model, such as OpenMP, UPC, CAF or Titanium, it is expected that such optimizations will be done by a compiler; the compiler analyzes the communication pattern entailed by the loads and stores performed by the program, and performs some of the previously discussed optimizations. Unfortunately, the experience of the last decades indicate that compilers frequently fail at doing a good job, especially for codes with dynamic, data dependent communication patterns. Furthermore, it is hard for a programmer to judge which memory access patterns are easy for a compiler to optimize and which are hard. A system where achieved performance can change significantly because of small, seemingly innocuous, code changes that throw off compiler analysis can be very frustrating.

Programmers can avoid the potential inefficiency of access to global data by explicitly performing the optimizations listed above: In a distributed memory system, users explicitly partition data and manage communication; in shared memory systems, they may copy global variables into local variables, to explicitly gather and cache global data. But, in doing so, the users have lost much of the advantage of using a global name space, namely avoiding the need for explicitly managing local copies. Furthermore, the copying is an unnecessary overhead on a machine with hardware supported coherent shared memory – an overhead that a compiler will have hard time optimizing away.

The alternative approach that we explore in PPL1 is to relieve the programmer from the burden of explicitly moving data and managing local copies; the user can program using global names to access variables. However, in order to overcome the limitations of compiler analysis, the user can provide information on the access patterns to data, more accurately or earlier than available from an analysis of program references. The information is provided as early as possible; if the memory access pattern is known at compile time, then the information can be provided declaratively, so that it be available to a static compiler; if the information stays unchanged for many successive iterations, then it may be provided outside a parallel loop, in support of run-time compilation.

The information provided essentially is information about the define-use relation in the parallel program: which thread updates a variable and which threads use the updated variable. Semantically, the information provided is redundant, and can be thought of as “annotation”; it does not affect the execution outcome, but merely ensures that information embedded in the program is made available to the compiler and run-time at an earlier point in time, thus possibly improving performance.

We describe in more detail the annotation mechanism provided to specify data access patterns in Section 11. The design faces several fundamental choices.

- Are annotations mandatory or optional?
- Do we use annotations only to specify certain data access patterns, or also to specify likely but not certain access patterns?
- Do incorrect annotations merely affect performance, or should an exception be generated if a program provides incorrect information on access patterns erroneous?

Making annotations mandatory clearly denies the goal of supporting an iterative refinement design methodology, where annotations are added only to improve performance as needed; on the other hand, mandatory annotations simplify implementation on distributed memory systems, since communication code can be generated from the annotations without analyzing accesses performed by the computation code.
PPL1 supports both views: the type of a variable indicates whether movement of that variable should be controlled by the programmer, or should be implied by the accesses to that variable. In the former case “data moving annotations” are mandatory and an attempt to access a variable at a site where it has not been moved to generates an exception.

We do not support, at present time, speculative data movement: Our intuition is that such an idiom would be seldom used.

### 2.3 Global Control

The basic view of parallel control that we advocate is that of a *loosely synchronous* (or bulk synchronous [22]) model: the computation is under sequential control; each “sequential step” is a phase where multiple threads, execute independently, with no interaction. The threads communicate and synchronize (logically) at the end of each phase.

The requirement for global synchronization may seem to be a significant restriction. But parallel numerical codes are often written in what is, essentially, a loosely synchronous style, where threads alternate between computation and communication phases. This model is often referred, informally, as the *compute-communicate* model.

One unappreciated feature of this model is that it is the parallel equivalent of sequential structured programming. In his famous short paper “Goto’s Considered Harmful” [8] Dijkstra argued against the use of goto’s, based on two arguments:

1. the progress of the execution of a structured program can be tracked using simple coordinates – loop indices, unlike programs that use goto’s; and
2. arbitrary programs can be transformed into equivalent structured programs with little loss of efficiency.

The same argument applies to a loosely synchronous program: the progress of a computation can be tracked by tracking the global progress through the sequential steps; and the progress of each individual execution thread since the last global synchronization; one need not worry about the relative order of operations at concurrent threads, since these do not interact and the order is immaterial. Furthermore, the work of Valiant [22] provides theoretical evidence that an arbitrary program can be transformed into a loosely synchronous program, with a constant factor increase in execution time and a logarithmic loss of parallelism.

In some cases, a pure loosely synchronous model may be too restrictive: it may be important to nest it. Consider the case of a multidisciplinary simulation; one may have coarse steps where different simulations interchange data; each simulation proceeds independently in between coarse steps. PPL1 supports arbitrary nesting of parallel constructs. The execution of a PPL1 program can be represented by a series-parallel graph; conflicting memory accesses are always ordered by this graph.

The use of global control and global name space results in a code that can be executed sequentially, with no modifications: parallel iterators can be interpreted as sequential iterators. The restrictions we impose on conflicting memory accesses imply that a sequential execution does not
change the outcome of the computation. This is a significant pragmatic advantage for code development: code can be developed and debugged in a sequential environment, next executed in parallel. The parallel execution will not introduce new bugs, but only performance tuning problems.

2.4 Virtual Processors

Libraries such as MPI or languages such as UPC or CAF provide a model of a fixed number of processes; both control and data are distributed across these processes. On the other hand, OpenMP provides a very dynamic view of execution threads that are spawned and allocated dynamically within parallel loop constructs; but OpenMP does not provide any control of data distribution or of communication. As a result, OpenMP codes often exhibit poor locality of reference.

The experience with Charm++ [18] and other similar systems show that processor virtualization has significant advantages and little performance cost, if any. By programming to virtual processors, the user expresses concurrency, but leaves to the system the details of process allocation. This flexibility is important in order to achieve modularity, as parallel programs are composed: A change in computational requirements of one subprogram may require reallocation of resources in another; with no virtualization, this needs to be done in the user code, so that encapsulation is lost. Virtualization also enables automatic load balancing, checkpointing and various communication optimizations.

The model of Charm++ and other similar systems is that a virtual process is a persistent entity that may be remapped to a different physical location at infrequent intervals. Thus, this entity can be used to manage locality; but unlike a system with no virtualization, one need not worry too much about load balancing, i.e., having each process perform the same amount of work in between synchronization points. PPL1 takes the same viewpoint, but goes one step further in making such “virtual locations” first class objects in the language. A locale is a virtual location that can support one thread of control and can carry data. Parallelism is obtained by mapping the execution of code onto a set of locales or a cohort; data is distributed by mapping it into a cohort; and locality of reference is achieved by mapping data and the execution that touches this data to the same locale.

A PPL1 program may use multiple cohorts, so that a different view of parallelism can be used for different subprograms or different phases of a computation. If these subprograms are independent, then this avoids the need to use a common frame of reference and enables the user to program in terms of the parallelism that is natural for the subprogram. Where threads or data sets need to be collocated, then the user can express this by collocating locales. The use of virtual locations as first class entities provides significant flexibility by separating data distribution from control distribution: a parallel execution can spawn only part of the locations containing data; conversely one can have multiple executing threads associated with the same data set.

2.5 Compatibility

Compatibility with MPI is provided by assuming the existence of a predefined cohort that corresponds to MPI_COMM_WORLD. PPL1 threads running on the locales of this cohort can collectively invoke a C or Fortran routine on each thread that can then use MPI for communication. Since PPL1 does not require the sandboxing mechanisms of Java, PPL1 private (local) only variables can be passed by reference from PPL1 to the C or Fortran library routine. If a standard implementation of MPI is used, this the locales of the “MPI_COMM_WORLD” cohort cannot be migrated, thus
restricting load balancing, since MPI processes would not be migratable. One can use an MPI implementation such as AMPI [17] that supports MPI process migration to avoid this limitation. In such a case, then it should be possible to provide an interface that enable to pass by reference partitions of distributed arrays, similar to the HPF extrinsic interface [16, Section 6].

We have not yet explored how PPL1 programs would be invoked from parallel program using MPI, OpenMP or UPC.

2.6 Portability

PPL1 should provide portability across a range of parallel architectures. It is designed to fit well systems that support efficient remote memory accesses via global shared memory hardware, intelligent adapters or communication co-processors [6, 3, 11]; it can take advantage of but does not require hardware support for coherent global shared memory. While PPL1 can be implemented atop a message passing layer, such implementation is likely to be less efficient.

Since PPL1 program can be executed on a sequential platform, it is entirely possible to develop a version of PPL1 where all parallel constructs are directives (or pragmas) that would be ignored by a conventional compiler. Instead, we present here a version of PPL1 where parallel constructs are part of the language. A simple preprocessor can translate PPL1 code into valid sequential code.

2.7 Dynamic vs. Static Parallelism

In a language such as OpenMP parallel control distribution is done dynamically, while data is not distributed; in a library such as MPI or a library such as UPC, data is statically (declaratively) distributed, and the set of execution sites and data storage sites is fixed. Compilation is easier with a fixed set of execution sites and with a static distribution of data; on the other hand, coding is easier in an environment that supports more dynamic data and control distribution and the dynamic creation of execution sites. PPL1 attempts to hit a middle ground: new execution sites can be created dynamically, and data can be redistributed; however a strict nesting discipline is applied: parallel control statements are well nested, and a parallel thread can only access data that was accessible at its parent, or that was locally instantiated.

Discussion

We may retreat into a more static design; in which case, we need to support “inter-cohort” communication and synchronization.

3 Java extensions for scientific computing

In order to properly support scientific computing in Java, we need to add new scalar types, such as Complex, and need to support assignments and arithmetic operators applied to aggregates, such as matrices; see, e.g., [13]. (As explained latter, PPL1 arrays are not the same as Java arrays – they are closer to the multiarrays proposed by the Numeric Java work group [19, 13].) We can either do it by supporting general operator overloading, or do it by adding new types to
the language and defining in the language the meaning of operators applied to operands of these types. In either case, we should be able to write something of the form \( a = b + c \times d \), rather than \( a \text{.assign}(b \text{.plus}(c \text{.times}(d))) \). The application of operators such as + or * to aggregate objects implicitly creates new object instances.

The initial implementation of PPL1 will probably extend arithmetic operators to handle specific newly added types, rather than support general operator overloading.

Overloading or extending the assignment operator = is a problem in Java, because of the way Java handles assignments to objects: If \( a \) and \( b \) are arrays then \( a=b \) assigns \( a \) to refer to the same array as \( b \); it does not copy the entries of \( b \). We need both types of behaviors: we need a shallow copy assignment operator that assign references, and a deep copy assignment operator that copies the entries of the aggregate (assuming that the aggregates on the lhs and on the rhs are “conforming”).

We use the operator := for the deep copying. The difference is illustrated in the code below.

```java
Matrix a = new Matrix(...);
Matrix b = new Matrix(...);
Matrix c = new Matrix(...);
a = b; \quad // a and b now refer to the same object
a := 1; \quad // the values of both a and b have changed;
\quad // both refer to a unit matrix
c := a; \quad // c has same value as a (but a and c refer to different objects)
c := c+2; \quad // the value of c has changed, but not the value of a
b = 3; \quad // illegal statement (3 does not refer to a matrix)
```

:= and = have the same semantic, when applied to scalars; the semantic of = is not changed. If we choose to support overloading, then := will be the operator that can be overloaded; if we choose to add native types, then := will be used to support deep copy semantics.

We also modify the Java Collection Framework [1] to suit our needs.

We borrow from Titanium [10, Chapter 5] the notion of an immutable class. Immutable classes are not extensions of any existing class (including Object), nor can they be extended. All non-static fields of immutable classes are final. These restrictions allow the compiler to pass such objects by value and to allocate them on the stack or within other objects. In effect, they behave like existing Java primitive types or C structs.

Note that if \( c \) refers to an instance of an immutable class object, then an assignment \( c = \ldots \) can be used to change the object \( c \) refers to. Logically, a new object instance is created; however, the compiler can do “in-place” update of the object instance, rather than allocating a new object instance.

---

**Discussion**

This section is preliminary, and will need additions and modifications. From Titanium or Fortress, we may borrow type constructors, atomic types, templates, operator overloading, etc.
4 Sets

In this section we extend and modify the sorted set interface of the Java Collection Framework [1]. The Java Collection Framework uses parameterized interfaces, with type parameters. Here and in the following sections, we replace the parameterized interfaces with concrete (non-parameterized) interfaces, defined for types of interest. This choice is made in order to facilitate the efficient implementation of the set methods with static compilation technology, and because the applications we consider do not seem to require the full generality of a parameterized interfaces. The design proposed here can easily be replaced with a design that uses parameterized interfaces, if compiler technology is available to optimize the concrete cases of interest.

A set is a collection of objects, all of the same type, with no duplications. A sorted set is a set with a total order defined on the elements.

PPL1 implement sorted sets with elements of each integral type and sets of points (points are introduced in Section 6). These sets have type, respectively, SetofByte, SetofShort, SetofInt, SetofLong, SetofChar and SetofPoint<N>, where N is a manifest integer. These are immutable classes. The set elements are ordered by the natural ordering of their type. These sets do not change once created.

We support the mandatory Java sorted set methods.

- EMPTY_SET is a constant representing the empty set.
- s.contains(a) returns true if set s contains element a.
- s.containsAll(t) returns true if set s contains all of the elements of set t.
- s.equals(t) returns true if sets s and t have the same elements.
- s.isEmpty() returns true if set s contains no elements.
- s.size() returns the number of elements in set s.
- s.toArray() Returns an array containing all of the elements in the set s
- s.first() returns the first element of s.
- s.last() returns the last element of s.
- s.subSet(a,b) returns the set of elements in s from a, inclusive, to b, exclusive.
- s.headSet(a) returns the set of elements in s that are smaller than a.
- s.tailSet(a) returns the set of elements in s that are larger of equal to a.

We can apply comparison operators to sets. Thus s < t evaluates to true if t properly contains s; s <= t evaluates to true if t contains s (same as t.containsAll(s)), s==t evaluates to true if the two sets have the same elements (same as s.equals(t), and so on.

PPL1 sets are immutable, and elements cannot be added or deleted from sets; however a new set can be created by adding or deleting elements from an existing set.
• \( s \text{.union}(t) \) returns the set of elements that are either in \( s \) or in \( t \).

• \( s \text{.intersection}(t) \) returns the set of elements that are both in \( s \) and in \( t \).

• \( s \text{.difference}(t) \) returns the set of elements that are in \( s \) but not in \( t \).

The same operations can be performed using the operators +, * and -: \( s+t \) evaluates to the union of the two sets; \( s*t \) evaluates to their intersection; and \( s-t \) evaluates to their difference.

An expression of the form \( \{a_1, a_2, ..., a_k\} \) where \( a_i \) is an expression of type \( T \) evaluates to a \( \text{Set}<T> \) object that contains the corresponding elements.

**Implementation Note**

A set is represented in PPL1 by a hash table that is used to find the index of an element in the set, and an array containing the elements in iteration order. Iterating over all the elements of a set will be as efficient as iterating over an array; finding a random element (invoking \( s \text{.contains}() \)) will incur the overhead of a search in a hash table, and may require a few random accesses to memory. Set operations and set comparisons take time linear in the set sizes. Invocations to \( s \text{.size}() \) and \( s \text{.isempty}() \) take constant time.

This implementation choice is biased toward sets that do not change, but are frequently iterated.

**Discussion**

Our main use for sets is as domains for maps. It is not obvious that we need general sets; it may be sufficient to support (or to support efficiently) only segments of integers and grids.

We could support multiple, distinct implementations for sets. Since our focus is on performance, it is important that the performance properties of types be as visible as their semantic properties. Thus, we do not propose to make the implementation choice transparent. We could provide user control of this choice, by using an implementation sublanguage [7, 2], or, less conveniently, by building a suitable class hierarchy. Both solutions, especially the first, require good compiler technology. We are not pursuing this direction for the time being.

### 4.1 Segments and Set Constructors

Sets containing all the elements between a lower bound and an upper bound are called *segments*. They have their own types: \( \text{SegmentofByte} \), \( \text{SegmentofShort} \), \( \text{SegmentofInt} \), \( \text{SegmentofLong} \), \( \text{SegmentofChar} \) and \( \text{SegmentofPoint<N>} \). All the methods defined on sets apply to segments; segments are implicitly coerced to sets when needed, and sets can be explicitly be coerced to segments. Segments have one constructor that accept two endpoint arguments. The expression \( \{lb..ub\} \) evaluates to the segment with endpoints \( lb \) and \( ub \). The expression \( \{..ub\} \) is synonymous with \( \{0..ub\} \).

We can also use the notation \( \{lb..ub\} \) in set defining expressions. Thus \( \{1..4, 7, 10..12\} \) denotes the set with elements 1, 2, 3, 4, 7, 10, 11, 12.
s.isSegment() returns true if the set s is a segment.

Discussion
An invocation to s.subset(a,b) returns all the elements that are greater or equal to a and strictly smaller than b; on the other hand, the set defined by the expression \{a..b\} contains b. This discrepancy may be misleading. We may deprecate the use of the subSet method, to prevent confusion.

PPL1 provides four standard set constructors:

- A constructor with no arguments constructs an empty set.
- A constructor with an array argument constructs the set containing the elements of the array.
- A constructor with a set argument constructs a copy of that set.
- A constructor with an element argument constructs a singleton set containing exactly that element.
- A constructor with arguments lb and ub constructs the segment consisting of all the elements that are greater or equal to lb and smaller or equal to ub.

Implementation Note
Segments of the types supported in PPL1 have a more concise representation than general sets, that takes constant space: one only needs to store the two endpoints. Operations on segments take constant time (except operations that create general sets, such as set union).

4.2 Set Operations

PPL1 provides a reduction operator that can be used to reduce elements of a set into a scalar value, for each associative binary operator defined in the Java language or the Java.lang.Math class. Thus, the following expressions can be used to “reduce” a set into a scalar value:

- s.and() returns the bitwise AND of the set elements (or the logical AND, if the set elements are boolean). s.all() is a synonym to s.and(). The and of an empty set returns an all 1 value.
- s.xor() returns the bitwise XOR of the set elements (or the logical xor, if the set elements are boolean). The xor of an empty set returns an all 0 value.
- s.or() compute the bitwise OR of the set elements (or the logical OR, if the set elements are boolean). s.any() is a synonym to s.or(). The or of an empty set returns an all 0 value.
- s.max() returns the maximum value of an element in the set. The maximum of an empty set equals MIN_VALUE.
• \texttt{s.min()} returns the minimum value of an element in the set. The minimum of an empty set equals \texttt{MAX\_VALUE}

• \texttt{s.sum()} returns the sum of the set elements. The sum of an empty set equals 0

• \texttt{s.product()} returns the product of the set elements. The product of an empty set equals 1.

The elements of the set may be reduced in arbitrary order, as it is assumed that the reduction operations are associative; for sets of floating point numbers, rounding errors may result in slightly different results.

\textbf{Discussion}

The reduction of an empty set returns the unit of the corresponding operation, so that reductions commute with set union: for example, \((s+t).\max\) equals to \texttt{max(s.max, t.max)}

Unary methods and operators can be applied, element-wise, to sets with elements of suitable type. For example:

• \texttt{!s} is a set containing the bit-wise complements of the elements in set \texttt{s}.

• \texttt{s.abs()} is a set containing the absolute values of the elements in \texttt{s}

We also support unary operators that are curried versions of binary operators. For example, \texttt{a+s}, where \texttt{a} is a scalar and \texttt{s} is a set, evaluates to the set containing all elements of the form \texttt{a+x}, where \texttt{x} is an element of \texttt{s}; \texttt{s+a} is similarly defined.

An element will be implicitly coerced into a singleton set containing this element, if it appears in a context where a set is required. Conversely, a singleton set can be explicitly converted to the contained element using a cast expression.

The operator \texttt{s.isSingleton()} returns \texttt{true} if \texttt{s} contains a unique element, \texttt{false}, otherwise.

\textbf{Missing Material}

May want to add further operators, such as set membership (\(\in\)).

Should add sets of \texttt{enums}.

Should add named methods for all operators.

Should decide whether to overload the cast operator: can we cast a set, by using the cast for the set elements?

\section{Maps}

We modify in this section the map interface of Java’s Collection Framework.
A map is an object of type `Map<X, Y>` that maps x-values (of type X) to y-values (of type Y). Each x-value is mapped to at most one y-value. The set of x-values mapped by M is the **Domain** of m and the set of y-values of m is the **Range** of M.

The domain of a map cannot be changed after the map was created. On the other hand, the y-values can be modified.

PPL1 implements maps where X has an integral type (byte, short, int, long or char), or a point type. are set types. These map classes are immutable.

PPL1 supports two constructors for maps:

- A constructor that takes a set argument and returns a map with this set as its domain, and all values in the range set to initial values (zero or null).
- A constructor that takes two array arguments and return a map constructed of pairs of elements from the two arrays.

PPL1 supports a map syntax similar to the Java array syntax.

- The expression `V[U]` defines the same type as the expression `Map<U, V>`.
- The expression `new V[s]`, where s is of type `SetofU`, creates a map of type `V[U]` with domain s.

Thus,

```java
float[char] a;
```

declares a to be a map from characters to float values. This declaration allocates the reference variable a and initializes it to null, but does not allocate space for the map itself.

If s is a set of type `SetofInt`, with elements 3, 5, 7, then the statement

```java
float[int] a = new float[s];
```

creates the variable a and initializes it to a map of three elements, with domain 3, 5, 7 and float range values that are initialized to zero.

Finally, one can initialize both the domain and the values of the map, using an initialization expression similar to that used in Java for arrays: Thus, if s is as before, then

```java
float[s] a = {1.3, 2.5, -5.7}
```

will create the map that associates 3 with 1.3, 5 with 2.5, and 7 with -5.7.

Java arrays
5.1 Map Methods and operations

Maps support the mandatory methods of the Java Map interface:

- `m.isinDomain(x)` returns `true` if `x` is in the domain of the map (same as `m.containsKey(x)`).
- `m.isinRange(y)` returns `true` if `y` is in the range of the map (same as `m.containsValue(y)`).
- `m.equals(n)` returns `true` if `m` and `n` represent the same map.
- `m.isEmpty()` returns `true` if the mapping contains no x-y pairs.
- `m.size()` returns the number of x-y pairs in the mapping.
- `m.Domain()` returns the domain set of the mapping (same as `m.keySet()`).
- `m.Range()` returns the range set of the mapping (same as `m.values()`).
- `m.get(x)` returns the range value to which `x` maps.

The composition of two maps, `m2.compose(m1)`, is the map that maps `x` to `z` if `m1` maps `x` to `y` and `m2` maps `y` to `z`. The composition is well defined if the domain of `m2` contains the range of `m1`. If `m1` is of type `Map<X,Y>` and `m2` is of type `Map<Y,Z>` then `m2.compose(m1)` has type `Map<X,Z>` and has the same domain as `m1`.

The inverse of a map, `m.inverse()`, is defined to be the map containing all pairs

\[ \{(x,y) : (y,x) \in m\} \]

This operation is well-defined only when `m` is one-to-one, i.e., only if each element in the range is the image of exactly one element in the domain.

**Implementation Note**
The standard representation of maps consists of a hash table that is used to find the index of an x-value, and two arrays of x-values and y-values. The standard implementation of maps ensures that `isinDomain` can be executed in constant time, and the value of a map for a particular domain value can be computed in constant time. On the other hand, `isinRange` may require a linear time search through all range values. Iteration over a map is as efficient as iteration over an array. Composition takes time linear in the size of the maps, while inversion has complexity $O(n \log n)$.

5.2 Map Access and Update

We are using array syntax for access to map values: Suppose that `m` maps `x` to `y`. Then the expression `m[x]` returns the value `y`. This is equivalent to an invocation of `m.get(x)`. An assignment of the form `m[x] := exp` changes the mapping by associating the value of `exp` with `x`.

Similar syntax can be used for accessing or updating more than one value in a map. The expression `m2[m1]` is synonymous with `m2.compose(m1)`. Therefore, `m2[m1]` is a map with the
same domain as \( m_1 \) so that \( m_2[m_1][x] = m_2[m_1[x]] \). In essence, the expression \( m_2[m_1] \) accesses the elements of \( m_2 \) with indices in the range of \( m_1 \) and index them using the domain of \( m_1 \).

For example, if \( m_1 \) is defined by the pairs \((1,a), (2,b), (3,c)\) and \((4,d)\), and \( m_2 \) is defined by the pairs \((0,2)\) and \((1,4)\) then \( m_1[m_2] \) is the mapping defined by the pairs \((0,b)\) and \((1,d)\).

Let \( m_1 \) be of type \( \text{Map}<Y,Z> \), let \( m_2 \) be of type \( \text{Map}<X,Y> \), and let \( m_3 \) be of type \( \text{Map}<X,Z> \), so that \( m_2 \) and \( m_3 \) have the same domain. Then an assignment of the form \( m_1[m_2] := m_3 \) replaces each \( y \)-value \( m_1[m_2[x]] \) with the value \( m_3[x] \). That is,

\[
m_1[m_2][x] = \begin{cases} 
m_3[x] & \text{if } x \text{ is in the domain of } m_2 \text{ (and } m_3) \\
m_1[x] & \text{otherwise} 
\end{cases}
\]

For example, if \( m_1 \) and \( m_2 \) are defined as in the previous example, and \( m_3 \) if the mapping defined by the pairs \((0,x), (1,y)\), then the assignment \( m_1[m_2] := m_3 \) updates \( m_1 \) to correspond to the pairs \((1,a), (2,x), (3,c), (4,y)\).

---

**Discussion**

The map access and update notation extends the subarray access and update notation used by Fortran90 and other array languages.

---

### 5.3 Conforming Map Operations

Two maps are *conforming* if they have identical domains. Operators that apply to the domain elements of maps are overloaded to apply, point-wise, to conforming maps. For example, if \( m, n \) and \( p \) are conforming maps with the same numeric type range, then the assignment

\[
p := m+n
\]

sets the value of \( p[x] \) to \( m[x]+n[x] \), for each \( x \) in the domain of \( p \); the assignment.

\[
p += m
\]

sets each entry \( p[x] \) to equal \( p[x]+m[x] \); the assignment

\[
p := -m
\]

sets each entry \( p[x] \) to equal \( -m[x] \); and so on. The operations can be applied to maps even if the ranges of the maps do not have the same type, using the usual Java type conversion rules for the range elements: for example, the assignment \( p := -m \) is valid if \( p \) and \( m \) are conforming, \( p \) ranges over \text{float}'s and \( m \) ranges over \text{int}'s.

If \( m \) and \( n \) are conforming maps with the same numeric type range, then the expression

\[
m < n
\]
results in a component-wise comparison of the entries in the range of \( m \) and \( n \), returning a map \( p \) with range elements of type `boolean` so that \( p[x] = (m[x] < n[x]) \); the same applies to all other comparison operators.

The methods in `java.lang.Math` are also overloaded to apply, component-wise, to maps, where it makes sense. Thus,

\[
p := \max(m, n)
\]

sets each entry \( p[x] \) to equal \( \max(m[x], n[x]) \); the assignment

\[
p := \text{abs}(m)
\]

sets each entry \( p[x] \) to equal \( \text{abs}(m[x]) \); and so on.

### 5.4 Map Reductions

For each set reduction method `foo` we have a corresponding map reduction method `foo` defined as \( m.\text{foo}() = (m.\text{Range}()).\text{foo}() \). I.e., the reduction of a map evaluates to the reduction of the range of the map.

We add the following two reduction operators:

- **\( m.\text{minloc}() \)** returns a domain element \( x \) so that \( m[x] = m.\text{min}() \). If the minimum is achieved at more than one location than the first such location in the domain is returned.

- **\( m.\text{maxloc}() \)** returns a domain element \( x \) so that \( m[x] = m.\text{max}() \). If the maximum is achieved at more than one location than the first such location in the domain is returned.

A map of type `Map<X,Y>` is implicitly coerced to a map of type `Map<X,Z>` if it appears in an environment where type `Map<X,Z>` is required, and \( Y \) can be implicitly coerced to \( Z \). Thus, an integer valued map will be implicitly converted to a float valued map, if needed, as shown in the example below:

```java
... int[{{[2,3]}]} a = new int[{{[2,3]}]};
float[{{[2,3]}]} b = new float[{{[2,3]}]};
float[{{[2,3]}]} c;
c := a+b;
```

array \( a \) is converted to an array of floats, and next added to array \( b \).

Converseley, explicit casts can be used to convert the domain values of a map.

In addition, a scalar will be converted to a conforming map with all values equal to this scalar, when it appears in a map operation, as shown below
... float[..[2,3]] a = new float[..[2,3]]; float[..[2,3]] b; b := a-3.5;

Each entry of array `a` is decreased by 3.5 and the result is stored in `c`: the scalar 3.5 is promoted to a conforming array, and then an array difference is computed.

---

**Missing Material**

Should add enumerated types as indices.

We should provide concrete interfaces for the Java map methods for domains and ranges of interest.

Should decide whether to overload type coercion. I.e., can we use `(int)m` to convert the range elements to `int`'s? Or do we insist on `(int[X])(m)`?

Should add name methods for the various operands.

---

### 6 Points

In this section we borrow and extend the design of the Titanium language [10] in order to add to Java tuples, called *points*.

A *point* is a fixed length tuple of integers; points or arity `N` (N-points) belong to the (immutable) class `Point<N>`. Points are used to index arrays 7 and locations 8.

An expression of the form `[i_1, \ldots, i_N]`, where `i_j` are integer expressions evaluates to an N-point object with the corresponding components.

Points can be viewed as mappings with a domain of the form `{1..n}` and range `int`\(^1\)\text{ range} \{0..n-1\} may be more consistent with C/C++/Java but `{1..n}` is more consistent with how dimensions are normally indexed.

Mapping notation can be used for component access: \(p[i]\) is the \(i\)-th component of point \(p\), for \(i=1,2,\ldots,n\).

The standard map expressions and methods can be applied to points. Thus,

- \(p.size()\) returns the arity of point \(p\); \(p.arity()\) is a synonym, that applies only to points.

- \(p[q]\) is the point obtained by composing \(p\) with \(q\). This is well defined point of type `Point<N>` if \(q\) has type `Point<M>`, \(p\) has type `Point<M>` and all the entries of \(q\) are in the range `{1..M}`. Thus, if \(p = [10,20,30]\) and \(q = [1,2,2,1]\) then \(p[q] = [10,20,20,10]\). Point composition can be used to permute the components of a point. We can also use the syntax \(p.compose(q)\).
The arithmetic operators +,-,*, and / can be applied component-wise to pairs of points of the same arity: \( p+q=[p[1]+q[1],...p[N]+q[N]] \); \( p*s=[p[1]*s,...,p[N]*s] \); etc.\(^2\) We can also apply operands such as \( \min \) or \( \max \)\(^3\).

The relational operators \(<\), \(>,\leq\) and \(\geq\) can be applied to pairs of points of the same arity, returning a map with boolean values. Thus \( (p < q).all() \) is true if \( p[i] < q[i] \), for all \( i \)\(^4\).

The following additional methods are defined. In the following, \((p\text{ and }q\text{ denotes }N\text{-points and }i, s, n\text{ denote integers.})\) Whenever a point operand is required in an expression, it can be replaced by scalar; the scalar \( s \) is coerced into a point \([s,...,s] \).

- the concatenation of two points is defined as follows: if \( p = [p[1],...,p[M]] \) and \( q=[q[1],...q[n]] \) then \( p&&q = [p[1],...,p[M],q[1],...,q[N]] \).
- projection: \( p\text{.project}(i) \) is the point \(<p[1],...,p[i-1],p[i+1],...,p[N]> \) obtained by deleting the \( i \)-th component of \( p \). More generally, if \( q \) is of type \( \text{Point}<M> \) then \( p\text{.project}(q) \) is the point of type \( \text{Point}<N-M> \) obtained by deleting from \( p \) the components with indices \( q[1],...,q[M] \).
- shuffle, unshuffle: \( p\text{.shuffle}() \) is equal to \(<p[1],p[[N/2+1]],p[2],p[[N/2+2]]...> \). Thus, if \( p \) is equal to \(<1,2,3,4,5,6,7> \) then \( p\text{.shuffle}() \) is equal to \(<1,3,2,4,5,6,7> \). \( p\text{.unshuffle}() \) is defined with the inverse permutation: it is equal to \(<p[1],p[3],...,p[[N/2]+2],p[2],p[4],...,p[N]> \). Thus, if \( p \) is equal to \(<1,2,3,4,5,6,7> \) then \( p\text{.unshuffle}() \) is equal to \(<1,3,5,2,4,6,7> \).
- \( p\text{.direction}(k) \) returns a point of arity \( n \) whose \(|k| \) component has value \( \text{sign}(k) \) and all other components are \( 0 \). Thus, if \( p \) is of type \( \text{Point}<5> \), then \( p\text{.direction}(3)=<0,0,1,0,0> \) and \( p\text{.direction}(-3)=<0,0,-1,0,0> \)\(^5\).

An array of integers \( A \) can be explicitly converted into a point, using the usual conversion syntax \((\text{Point}<N>) A \)\(^7\). Also, an \textit{int} value is implicitly coerced into a \textit{Point}<1> value when needed, and vice-versa.

### 6.1 Sets of Points

Set of points have type \( \text{setofPoint}<N> \), where \( N \) is a manifest integer. Sets of points are ordered sets, with points ordered in lexicographic order: \( p \) precedes \( q \) if \( p[i] = q[i] \) for \( i=1...k-1 \) and \( p[k]<q[k] \).

Regular set operations can be applied. In particular:

---

\(^2\)Titanium defines division to round toward \(-\infty\), rather than \(0\). Need to check why.
\(^3\)It would be useful to support \( \min \) or \( \max \) with more than two arguments.
\(^4\)Titanium defines \( p\text{q} \) the way we define \( (p\text{q}).all() \).
\(^5\)If points are typed by their arity, then we cannot have a projection operator where the number of dimensions deleted depends on the value of the projection argument (rather than its type). Should check what breaks if \( N \) is not part of the point type.
\(^6\)Titanium allows an extra argument for the vector norm; also, it is more logic to pass the point type as argument, rather than a point.
\(^7\)Alternatively, we can have an explicit point constructor: \( \text{Point}<N>.\text{toPoint}(A) \).
• $s.arity()$ returns the arity of the points in set $s$.

• $s.compose(q)$ consists of all points $p.compose(q)$, for $p$ in $s$.

• $s + q$ is the set containing all points $p + q$, for $p$ in $s$; similarly for $p + s$, and for other operators that apply to points.

• $s.project(i)$ “projects” set $s$ along the $i$-th dimension: It returns the set containing all points $p.project(i)$, for $p$ in $s$. $s.project(p)$, where $p$ is a point, is similarly defined.

• $s && t$ is the set obtained from the concatenation of every point in $s$ with every point in $t$.

We add a few new operators:

• $s.lb()$ returns a point $p$ such that, for each $i$, $p[i] = \min\{q[i] : q \in s\}$; $s.lb(i)$ returns $p[i]$. $s.ub()$ and $s.ub(i)$ are similarly defined, with $\min$ substituted with $\max$.

• $g.extent()$ returns a point with entries that are the extent of the set in each dimension; if $g$ is defined as in 1 then $g.extent()$ returns the point $[ub_1 - lb_1, \ldots, ub_k - lb_k]$. $g.extent(i)$ returns the extent in the $i$-th dimension; it is the same as $g.extent()[i]$.

$s.lb$ returns the “lower-left” corner of the box occupied by the points in set $s$, and $s.max()$ returns the “upper-right” corner. Note that this is different than $s.min()$ and $s.max()$ that return the lexicographically first and last points in the set.

**Implementation Note**
The representation for sets of points is augmented to enable efficient iteration through projections. For 2-points, this corresponds to the standard, row-major representation for sparse arrays:

Let $\lambda > 0$ be a fixed threshold. We say that the set $s$ of 2-points is dense if $s.size() \geq \lambda(s.ub(1) - s.lb(1))$. I.e., the set is dense, if the average number of points in each row is $> \lambda$. If the set is dense, then we add a row list that points to the first point in each row. This will optimize iterations over the elements of a row (these elements are stored contiguously).

### 6.2 Grids

A **grid** is a set of points obtained by concatenating segments of integers (i.e., of `Point<i>` values). We use `Grid<N>` for the type of grids with points of arity $<N>$.

If grid $g$ is defined as

$$
\{lb_1..ub_1\} \&\& \ldots \&\& \{lb_k..ub_k\}
$$

then $g$ consists of all the points $p$ or arity $k$ so that $lb_i \leq p[i] \leq ub_i$, for $i = 1, \ldots, k$. The grid is a Cartesian mesh. For example, the grid

$$
\{2..5\} \&\& \{3..4\}
$$

Grids will be used to index dense multidimensional arrays.

The method

\( s \text{.isGrid()} \)

where \( s \) is a set of points, returns \texttt{true} if \( s \) can be converted to a grid; \( s \text{.isGrid()} \) will return \texttt{true} if \( s \) contains all points \( p \) such \( s \text{.lb}() \leq p \leq s \text{.ub}() \). Set of points of the right shape can be explicitly coerced into grids and grids are implicitly coerces to set of points, when needed. All operators that apply to sets of points apply to grids.

Note that if \( s \) is a grid then \( s \text{.lb}() = s \text{.min}() \) and \( s \text{.ub}() = s \text{.max}() \). We thus define the standard grid constructor to take two point arguments \( p \) and \( q \) and to return the grid \( g \) so that \( g \text{.min}() == p \) and \( g \text{.max}() == q \).

- \( g \text{.accrete}(k, \text{dir}) \) extends the grid \( g \) by \( k \) elements in the direction \( \text{dir} \). This is the same as \( g + k \times p \text{.direction}(\text{dir}) \).
- \( g \text{.accrete}(k) \) expands the grid \( g \) is all dimensions by \( k \) elements; \( g \text{.accrete()} \) is the same as \( g \text{.accrete}(1) \).
- \( g \text{.shrink}(k, \text{dir}) \) shrinks the grid \( g \) by \( k \) elements in the direction \( \text{dir} \). I.e., it returns the grid consisting of all points in the intersection of the sets \( g, g - p \text{.direction}(\text{dir}), \ldots, g - k \times p \text{.direction}(\text{dir}) \).
- \( g \text{.shrink}(k) \) shrinks \( g \) in all directions; \( g \text{.shrink()} \), is the same as \( g \text{.shrink}(1) \).
- \( g \text{.border}(\text{dir}) \) is the grid that contains the \textit{boundary} or grid \( g \) in direction \( \text{dir} \); more generally, \( g \text{.border}(k, \text{dir}) \) contains the the \( k \)-boundary (a boundary of thickness \( k \) of \( g \) in direction \( \text{dir} \); it consists of all points \( p \) in \( g \) so that \( p + i \times q \text{.direction}(\text{dir}) \) is not in \( g \), for some \( 0 < i \leq k \).

The shrink and accrete operations are illustrated in Figure 6.2.

**Implementation Note**

A grid is represented by a pair of points; all grid operations that yield grids can be computed in time linear in the arity of the grid points.

---

8Titanium allows an extra argument.
Missing Material
Need to discuss how one represents sparse sets of points of more than 2 dimensions.

Need to add examples.

7 Arrays

Arrays (also called Java arrays) can be taken to be maps with a domain that is a segment of integers, starting at 0. The expression T[], where T is a type expression, represents the type of arrays with elements of type T. Arrays can be declared and initialized using the usual Java syntax.

Extended Arrays (called, for short, arrays, when there is no risk of confusion), are maps with a domain that is a grid. An extended array with a domain of N-points is a rectangular, N-dimensional array. The expression T[N], where T is a type expression and N is a manifest integer, represents the type of extended arrays with domain of type Grid<N> and range values of type T. Domain elements of arrays or extended arrays are also called indices.

Arrays are implicitly coerced to extended arrays and extended arrays are implicitly coerced to maps when needed. Explicit type coercion can be used in the other direction.

In addition, we support implicit type coercion from sets to arrays. Thus, if the expression {3.5, 0.0, 2.1} appears where an array of floats is needed, it will be converted into a Java array of floats with three values, indexed from zero.

Note that type T[2] is not the same as T[][] . An array of type T[2] is rectangular; once the array is allocated, it does not change size or shape.

Implementation Note
Extended arrays are represented by a pair of entities: a “dope vector” that holds the grid starting indices and extents and the start address of the area that stores the value; and a contiguous memory range that stores the array values; the values are stored in row-major order. Both the dope vector and the memory range storing the elements are cache line aligned.

7.1 Array Creation

Arrays (and extended arrays) can be declared and initialized using the same notation as for maps; we use array creation syntax, rather than constructor invocation syntax.

The declaration

float[2] a;

declares a to be a (reference to a) 2-dimensional extended array. The declaration allocates the variable a and set it to null but does not allocate space for the array.

23
The statement

```plaintext
a = new float({[-2,0]..[3,2]});
```
allocates space for a $6 \times 3$ array of floats (and its dope vector) and initializes it to zero; $a$ now refers to this array. The domain of this array is $\{-2..3\} \& \{0..2\}$ The statement is equivalent to the following code

```plaintext
g = new grid([-2,0], [3,2]);
a = new float[g];
```

The statement

```plaintext
int[2] m = {{2, 3, 5},{-2, 0, 1}}
```

Creates a $3 \times 2$ array initialized to the values shown; the syntax for the array initializer is the same as in Java (except that all subarrays must have the same size and shape). The newly created array will be indexed from zero, as if created by the declaration $m = \text{new int}[0..3, 0..2]$;.

---

**Discussion**

We allow, like FORTRAN90, and unlike C and Java, a nonzero lower bound for array indices. This adds an extra cost to indexing operations (probably negligible, on modern architectures); need to decide whether users care about the extra functionality.

---

### 7.2 Array Access

Arrays are accessed or updated using the same notation as for maps, as shown in the example below

```plaintext
int a[2] = new int[\{..[2,3]\}]; // creates 3x4 array
int b[] = new int[0..3];
Point<2> p = [1,1];
int i;

a[[1,1]] := 5; // assignment to one array entry
a[p] := 3; // same entry is reassigned
i := a[p]; // access to an array element
b := a[{1..2}&&{1..2}]; // same as $b[0] = a[[1,1]]; b[1] = a[[1,2]]; b[2] = a[[2,1]]; b[3] = a[[2,2]]$

a[{0..2}&&1] := {7, 2, -1}; // same as $a[[0,1]] = 7; a[[1,1]] = 2; a[[2,1]] = -1$
```

Consider the last two assignments: First, the assignment

---

9I do not see how to have both an array initializer and a non-zero lower bound, without violating the syntax logic.

---
b := a[1..2&&1..2];

The expression \( \{1..2\}\&\&\{1..2\} \) produces the grid with points \([1,1], [1,2], [2,1] \) and \([2,2] \). This set is coerced into a one dimensional array with indices 0..3 and four Point<2> values. The two arrays are composed, returning a one dimensional array with indices 0..3 and values a[1,1], a[1,2], a[2,1] and a[2,2]. Array b is now updated to hold these four values.

Next, the assignment

\[
a[\{0..2\}\&\&1] := \{7, 2, -1\}
\]

The expression \( \{0..2\}\&\&1 \) evaluates to the grid containing the point \([0,1], [1,1] \) and \([2,1] \). This set is coerced into an array (map) that with indices 0, 1 and 2 and these three points as values. The set on the right hand side is also coerced into an array with indices 0, 1 and 2 and values 7, 2 and -1. We now have a map assignment that assigns to the entries b[0,1], b[1,1] and b[2,1] the values 7, 2 and -1, respectively.\(^{10}\)

### 7.3 Array Operations

All the methods that apply to maps apply to arrays as well. In addition, query functions that apply to grids can be applied to array; they return the information on the index set of the array. Thus, if a is an array, then

- \( \text{a.arity()} \) returns the number of dimensions in the array.
- \( \text{a.lb()} \) returns the (lexicographic) first index of the array. (On the other hand a.min() operates on the array values.)
- \( \text{a.ub()} \) returns the (lexicographic) last index of the array.
- \( \text{a.extent()} \) returns the extent of the array in each dimension.

As for maps, we can use standard operators to support element-wise array operations. Thus, if a, b and c are conforming arrays then then the assignment

\[
a := b+3*c
\]

will assign to each element \( \text{a}[[i,j]] \) the value \( \text{b}[[i,j]]+3*\text{c}[[i,j]] \). An exception will be raised if the arrays are not conforming. The assignment

\[
a := \text{min}(0, b)
\]

will assign to each element \( \text{a}[[i,j]] \) the value \( \text{min}(0, \text{b}[[i,j]]) \).

---

\(^{10}\)We could simplify syntax, by allowing \( a[1,1] \) instead of \( a[[1,1]] \); should do so if this does not introduce ambiguities.
Reduction operations are also supported, as for sets and maps. Thus, if \( a \) is an array of \texttt{float} and \( b \) is an array of \texttt{boolean}, then \( a \cdot \text{sum}() \) returns the sum of all entries in \( a \) and \( b \cdot \text{all}() \) returns the logical AND of all entries in \( b \).

For arrays, it is also convenient to support partial reductions on one dimension. Each of the reduction operations accept an additional dimension argument, indicating that the reduction occurs in that dimension, as shown in the example below.

```java
int[2] a = {{2, 3, 5},{-2, 0, 1}}
b = new int[3];
c = new int[2];
b := a \cdot \text{sum}(2);  \text{\// reduce in second dimension;}
    \text{\// sets b to \{0, 3, 6\}}
c := a \cdot \text{min}(1);  \text{\// reduce in first dimension;}
    \text{\// sets c to \{2, -2\}}
```

Note that \( a \cdot \text{sum}(i) \) returns an array of one dimension less than \( a \), with index set \((a \cdot \text{domain}()) \cdot \text{project}(i)\).

**Discussion**

Fortran also support, for many of the array operations, a \texttt{mask} argument (array of \texttt{boolean}). We can add it as an additional optional argument.

Several methods are specific to arrays.

- \( a \cdot \text{merge}(b, \text{mask}) \) merges arrays \( a \) and \( b \) under the control of the boolean array \( \text{mask} \). All three arrays must be conforming, and \( a \) and \( b \) must be of the same type. The resulting array is also conforming and associates index \( p \) with value \( \text{mask}[p] \ ? a[p] : b[p] \).

**Missing Material**

Needed to provide the equivalent of Fortran \texttt{PACK}, \texttt{UNPACK}, \texttt{MATMUL}, \texttt{RESHAPE}, \texttt{TRANSPOSE}, \texttt{CSHIFT}, and \texttt{EOSHIFT}.

**8 Sites**

A \textit{site} is a virtual location that can carry one thread of control and can have data associated with it: local data that is associated permanently, or global data that can be \textit{cached} at the site. The run-time manages the physical location of sites: sites can migrate from processor to processor, under the control of the run-time or under user control, in order to balance load or reduce communication.

Attributes can be associated with sites, in order to provide information about their properties. For example, a site can be “hardwired” to a particular processor or to a particular node. Such \textit{pinned} sites are not necessary for regular user codes, but may be needed for backward compatibility with MPI codes or for housekeeping functions. In a heterogeneous system, in particular in a grid system, it is possible to specify that a site should be associated only with certain types of nodes.

26
or with certain locations. In order to support out-of-core algorithms, one can have a site that is associated with secondary storage; thus migration of data to and from secondary storage can be handled using the same notation as for migration of data across a distributed memory system.

**Discussion**

We could associate attributes with cohorts, rather than sites.

Sites can be *collocated*, to indicate that they need to be at the same physical location. This can be useful for managing locality, i.e., ensuring that a thread is close to the data it uses. Conversely, they can be *anti-located*, to ensure that they are never at the same location. This can be useful for fault tolerance.

Sites are objects of the class `Site`.

A *cohort* is an indexed set of sites, i.e. a one-to-one map with a range values of type `Site`.

Note that a cohort does not have a “topology”: there is no notion of distance between sites. We assume that the physical topology is handled by the run-time, not by the user.

Cohorts are created by map (or array) constructors; when such a map is created, then each entry in the range of the map is initialized to a distinct, new site.

The method `mysite()` returns a reference to the site where the invocation occurs.

**Discussion**

We could define a cohort as a set of sites, but it is convenient to have indices, and to have flexibility on how sites are indexed.

### 8.1 Site Properties

A list of *properties* that consist of key-value pairs is attached to each site; both keys and values are strings. The meaning of properties is implementation dependent. Such properties can be used by the programmer to indicate the desired location of sites, or to inquire about the location of sites. In particular, attributes can be used to collocate sites, or to associate them with a particular physical resource (processor, node or IO device).

We use an interface similar to the one used by the Java class `Java.util.Properties` for the property related methods of the class `Site`. The list of keys associated with a site is specified by the site constructor. In addition, an implementation may have a default list of keys that is associated with each site.

- `s.getProperty(key)` returns the property attached to `key`. An exception occurs if no such key exists.
- `s.setProperty(key, value)` associates `key` with property `value`. An exception occurs if `key` is not settable or is not on the list.
• `s.isSettable(key)` returns `true` if the key can be reset. An exception occurs if there is no such key.

• `s.propertyNames()` returns a set containing all keys in the property list.

An implementation may further extend this class and associate various actions (such as site migration) with attribute updates.

PPL1 provides a cohort constructor that takes as argument a map into lists of **key-value** pairs and return a cohort with sites having the specified attribute values.

---

**Discussion**

A Side-effect of such constructor will be to locate or co-locate sites as desired.

---

### 8.2 Predefined Properties

The following property keys are predefined.

**Location** A key identifying the location where the thread executes (a URI).

**Persistent** Equals to “1” if the site is persistent, and “0”, otherwise.

---

**Missing Material**

Need to define the precise syntax for cohort constructors.

---

### 9 Parallel Control

#### 9.1 Forall

The basic construct for parallel execution is a **forall** statement. This statement is syntactically similar to the enhanced **for** statement of Java [15, 14.14.2]. We recall that that statement can be used to iterate over an array or a collection. The syntax of a **forall** statement is:

```
ForeachStatement:
forall ( VariableModifiers_opt Type Identifier: SetExpression; OnPart WherePart_opt) Statement
  OnPart:
on CohortExpression [ IndexExpression ];
  WherePart:
  ; where WhereExpression
```

---

28
The `SetExpression` must evaluate to a set of type `Set<Type>`. The loop body is executed once for each element of the set, with `Identifier` set to this element. The iterations execute concurrently. Each iterate has its own local copy of `Identifier`, initialized to a different initial value.

The `OnPart` specifies the site where the iteration is executed.

`WhereExpression` should evaluate to a `boolean` value; the iterate executes only if the expression evaluates to `true`. If `WhereExpression` is missing then it is assumed to have the default value `true`.

**Example:** The following program declares an array and initializes it to be a unit matrix; all entries are set in parallel.

```java
global int A[2] = new int[..[99,99]] \// declare a 100x100 integer array
  \// declare a 100x100 array of sites
forall (Point<2> p : ..[99,99]; on S[p])
  if (p[1]==p[2])
    then A[p[1],p[2]] = 1;
  else A[p[1],p[2]] = 0;
```

The `global` modifier is explained in Section 11.

Rather then setting up all the entries, it is sufficient to set up the diagonal entries. Only 100 sites are needed. The modified program is shown below.

```java
global int A[2] = new int[..[99,99]] \// declare a 100x100 integer array
site S[2] = new site[..99];
  \// declare a 100 array of sites
forall (Point<2> p : ..[99]; where p[1]==p[2]; on S[p[1]])
  A[p[1],p[2]] = 1;
```

A good compiler should implement this `forall` loop so that it is roughly as efficient as

```java
site S[] = new site[100];
forall (int i : ..99; on S[i])
  A[i,i] = 1;
```

**Discussion**

The enhanced `for` statement in Java would iterate through an array by iterating through the `elements` of the array; the `forall` loop iterates through the `indices` of the array. The former choice is preferable when the array is treated as a collection of elements that stand in no relation to each other, and each element is processed independently. In scientific computing, the location of an element in the array is meaningful: e.g., in an iterative algorithm, one updates the value of an element using the values of its neighbors; the “neighborhood” relation is defined by the index
values. In such a situation, it is more natural to iterate through the index set.

---

**Implementation Note**

A generic implementation of the `forall` loop will have each site execute a loop of the form

```plaintext
for (i : set)
  if (on(i) == mysite()) && where(i)) then statement;
```

A compiler will hoist the two tests out of the loop, so that the loop is executed only for the iterations that satisfy the conditions, whenever possible. A good compiler should issue a warning if it cannot perform this transformation.\(^{11}\)

---

### 9.2 Parallel

The `parallel` construct is used to spawn parallel execution of distinct statements. The syntax is

```
ParallelStatement:
parallel { ParallelStatementGroups }
  ParallelStatementGroups:
    ParallelStatementGroup
    ParallelStatementGroups ParallelStatementGroup
    ParallelStatementGroup:
on SiteExpression : BlockStatements
```

Each `SiteExpression` should evaluate to a distinct site.

**Example:** The following program sets in parallel the diagonal entries of a 3x3 matrix to one.

```plaintext
site S[] = new site[3];
parallel {
  on S[0]: A[1,1] = 1;
  on S[1]: A[2,2] = 1;
  on S[2]: A[3,3] = 1;
}
```

Note that the sites appearing in a `parallel` statement need not belong to the same cohort; on the other hand, all instances of a `forall` loop execute on sites of the same cohort.

**Discussion**

\(^{11}\)Should provide guidance to users about syntax for "easy" `on` and `where` expressions that can be "inverted" and hoisted out of the loop.
By restricting `forall` statements to span one cohort one simplifies the implementation of barrier statements (those are introduced below): a barrier synchronizes either a full cohort or a small set of named sites.

9.3 Abrupt Completion

`continue` and `break` statements can be used to transfer control within a thread, but cannot be executed by one thread so as to affect control in another thread.

A `continue` statement cannot be used to transfer control from inside a parallel block (`forall` or `parallel`) to a statement enclosing the parallel block or to the `forall` or `parallel` enclosing statements. It can be used to transfer control within the body of the parallel statement. Thus, the program

```
...  
forall(...)  
  while(exp) {  
    stat1;  
    if (cond) then continue;  
    stat2;  
  }
```

is correct; the `continue` statement abruptly completes the execution of one instance of the `while` loop.

While the program

```
...  
forall(...)  
  while(exp) {  
    stat1;  
    if (cond) then continue;  
    stat2;  
  }
```

is incorrect, and a compile-time error will occur.

An unlabeled `break` statement can be used to transfer control inside an instance of a parallel block, or to complete the execution of an instance of a parallel block; but a labeled `break` cannot be used inside a parallel block to transfer control outside the block. Thus

```
...  
forall(...)  
  while(exp) {  
    stat1;  
    if (cond) then break;  
    stat2;  
  }
```
is correct; the \texttt{break} statement abruptly completes the execution of the \texttt{while} statement.

The program

\begin{verbatim}
... 
forall(...){
    stat1;
    if (cond) then break;
    stat2;
}
\end{verbatim}

is also correct; the \texttt{break} statement completes the execution of the body instance where it is executed, but does not affect the other instances.

The program

\begin{verbatim}
id: stat0;
forall(...){
    stat1;
    if (cond) then break id;
    stat2;
}
\end{verbatim}

is incorrect and a compile-time error will occur.

\section{9.4 Barriers}

A \texttt{barrier} statement can be used to synchronize concurrent threads.

The syntax of a simple barrier invocation is

\begin{verbatim}
SimpleBarrierInvocation:
    barrier()
\end{verbatim}

When a thread reaches a simple barrier invocation then the thread blocks until all other threads of the innermost parallel construct containing the thread reached a simple barrier invocation. All threads then resume execution.

Logically, the execution of a code of the form

\begin{verbatim}
forall(forall_header) {
    part1
    barrier();
    part2
}
\end{verbatim}

is equivalent to
Note, however, that the association of barrier statements from different threads with one another is dynamic and may depend on the program control flow. In the example below

```c
forall(forall_header) {
    part1
}
forall(forall_header) {
    part2
}
```

Some threads may execute the `then` statement, while other threads may execute the `else` statement; the barrier will execute correctly since all threads will hit a barrier statement.

The full barrier syntax can be used to associate the execution of statement block with the barrier invocation. The syntax is shown below:

`SyncStatement:`
- `sync` `ConcurrentStatement` `Barriers`

`ConcurrentStatement:`
- `ForeachStatement`
- `ParallelStatement`

`Barriers:`
- `BarrierClause`
- `Barriers BarrierClause`

`BarrierClause:`
- `BarrierLabel : Block`
- `BarrierLabel: case ConstantExpression :`
  - `case EnumConstantName :`
  - `default:`

The overall syntax is similar to the `try-catch` syntax of Java; the syntax of the `barriers` part is similar to the syntax of a `case` statement in Java.

The barrier invocation syntax is

`BarrierInvocation:`
- `barrier()`
- `barrier(ConstantExpression)`
- `barrier(EnumConstantName)`
When a thread reaches a barrier invocation than it blocks until all the other threads within the same innermost parallel construct reach a barrier invocation or complete. If a thread invoked a barrier with no argument while another invoked the barrier with an argument, or two threads invoked their barrier with arguments that evaluate to distinct values, or one thread invoked a barrier while another completed then a *deadlock* exception occurs. Otherwise, control is passed to the parent thread – the thread that executed the parallel construct. If all the threads invoked the barrier with no argument and there is a barrier statement labeled with `default` then the associated block statement is executed. If all threads invoked the barrier with an arguments that evaluated to the same value and there is a barrier statement labeled with that value, then the associated block statement is executed. In both cases, each thread resumes execution immediately after the barrier. If no matching barrier label is found then an exception occurs.

Consider the following example:

```java
global int i, j, sum;
Site[] c = new Site[2];
sync {
    parallel {
        on c[0] : {
            i = 3;
            barrier();
            i = sum;
        }
        on c[1] : {
            j = 7;
            barrier();
            j = sum;
        }
        default: sum = i+j;
    }
}
```

At the exit from the parallel statement all three variables `i`, `j` and `sum` have value 10. Note that the statement `sum = i+j` is executed only once.

### 9.5 Nesting

Parallel constructs can be nested. However, a run-time error will occur if two possibly concurrent threads are associated with the same site. This is defined more formally in Section 9.6.

The next program is legal:

```java
Site s[] = new Site[100];
forall (int i: {0..99}; on s[i]) {
    Site t[] = new site[i];
    forall (int j: {0..i}; on t[j]) stat;
}
```
This program creates a cohort with 100 sites, next creates 100 cohorts with a total of 5050 distinct sites, for a total of 5150 sites.

**Discussion**
While this is a legal program, it is likely to be an inefficient one: we expect that the creation of a cohorts will be an expensive operation, so that users should create a site only for reasonably large tasks.

The following program is also legal:

```java
global Site[] s = new Site[30];
parallel {
    on s[0]: {
        stat1;
        forall (int i : {0..9}; on s[i]) stat2;
    }
    on s[10]: {
        stat3;
        forall (int i : {10..29}; on s[i]) stat4;
    }
}
```

Note that site $s[0]$ spans a parallel loop with one iterate executing at site $s[0]$.

The following program is illegal:

```java
global Site[] s = new Site[30];
parallel {
    on s[0]: {
        stat1;
        forall (int i : {0..9}; on s[i]) stat2;
    }
    on s[10]: {
        stat3;
        forall (int i : {9..29}; on s[i]) stat4;
    }
}
```

The program schedule two execution threads concurrently on site $s[9]$, hence will result in a run-time error.

Barriers can also be nested, in effect “undoing” the nesting of parallel statements. This is illustrated in the following example:

```java
... int a;
```
Site[] s = new Site[10];
sync {
    forall(int i : {0..9}; on s[i]) {
        int b = i;
        Site[] t = new site[5];
        sync {
            forall (int j : {0..4}; on t[j]) {
                int c = j; \ \ executed on an instance of t[j] that was spawned by s[i]
                barrier() \ \ the barrier synchronizes one instance of
                        \ \ cohort t[0]...t[4]; it executes on s[i] and assigns
                        \ \ value 5 to one instance of b; next it
                        \ \ invokes a nested barrier that executes on
                        \ \ main and assigns value 13 to a
            }
            default: {
                b = 5;
                barrier();
            }
        }
        default: a=13;
    }
}

9.6 Order

The semantics of Java impose a total order on the actions that occur during program execution, assuming that threads are not used; we call this order program order. This order is extended to a partial order on the actions that occur during the execution of a PPL1 program, using the following additional rules:

1. All actions due to the execution of instructions that precede a concurrent statement complete before any action due to instructions in the concurrent statement starts.

2. All actions due to instructions in a concurrent statement complete before any action due to the execution of instructions following the concurrent statement start.

3. All actions due to the execution of instructions preceding the execution of a barrier complete before any action due to the execution of the barrier clause starts.

4. All actions due to the execution of a barrier clause complete before any action due to the execution of an instructions following the barrier invocation starts.

The resulting order is represented by a series-parallel graph.

Two statements $s_1$ and $s_2$ are definitely ordered if a flow analysis of the code can determine that an instance of $s_1$ is never concurrent with an instance of $s_2$; $s_1$ and $s_2$ are possibly concurrent, otherwise. The flow analysis is done ignoring nonconstant values; for example, whenever a branch depends on a nonconstant condition, the assumption is that the branch can go either way. Consider, for example, the following example
int n=1; Site s[] = new Site[n];
forall (int i : 0; on s[i]) statement;

Then statement is possibly concurrent with itself; this, even though the parallel loop has only one iteration: Static flow analysis does not consider the values of non constant expressions (in this case, the OnExpression), and will assume that there may be more than one iterate executing concurrently. Similarly, in the following example,

... site s[] = site S[100];
forall (int i : 0; on s[i])
if (i==0) then x=3;

then then assignment x=3 is possibly concurrent with itself, as the static flow analysis cannot determine that it is executed only on one site.

Finally, in the example below,

global int a; ... Site s[] = new Site[2]; parallel on s[0]: if a == 0 statement1; on s[1]: i

then statement1 is possibly concurrent with statement2.

By extension, we say that two sites are possibly concurrent when they execute possibly concurrent statements.

Implementation Note
A static compiler analysis will compute the “possibly concurrent” relation for statements in a PPL1 program. A run-time check will raise an exception if two possibly concurrent statements are assigned to the same site. Note that the site assignment may not be known at compile time.

Discussion
We could also have default rules for changing the access modes of variables when a parallel section starts; e.g., from exclusive to shared. It seems preferable (at least initially) to keep things simple.

Missing Material
Need more examples and illustrations.

Need to further refine and formalize the “possibly concurrent” relation.

Need to speak of exceptions that are not caught within the thread that raises the exception: what is the state after such an exception. Probably need a standard exception “uncaught exception in parallel thread”.

37
10 Atomic blocks and Reductions

A problematic issue with shared memory languages is the handling conflicting accesses to shared variables. Conflicting accesses introduce nondeterminacy, resulting in hard to understand and hard to debug code. Nonatomic accesses to shared variables result in undesirable, hardware dependent outcomes. For example, Java does not guarantee that an access or an update to a 64 bit value is atomic; if one thread reads a double precision number while another concurrently writes this number, then it is possible that the value returned by the read has 32 bits from before the write and 32 bit from after the write, which is a meaningless answer. On the other hand, if atomicity has to be supported in software using a locking protocol, then one may suffer significant overheads.

Conflicting shared memory accesses are often the result of sloppy programming or of the lack of support for suitable synchronization mechanisms. We see three main reasons for the use of concurrent shared memory accesses.

- A reactive application handles external events that may occur (nearly) concurrently; “event handlers” that handle the occurrence of such events may need to execute concurrently (to provide quality of service) and may need to concurrently update shared state. The prototypical example of such a system is an Online Transaction Processing System; database transactions have to be atomic.

- One wishes to process in parallel a data structure, such as a graph, where aliasing is inherent in the data structure: a graph node can be reached through different edges, so has, inherently, multiple, distinct references; the same node can be accessed concurrently by different threads. A prototypical example of such a system is a parallel graph traversal that marks reached nodes; accesses to node structures have to be atomic.

- One wishes to update in parallel a single variable, e.g., to accumulate a sum or a count. Such reductions are (or are assumed to be) commutative and associative, so that the order of execution of the updates is immaterial; all participating threads use the same reference.

PPL1 supports atomic blocks as a general mechanism for atomic access to shared memory. The syntax of an atomic section is

atomic Block

The atomic block should not contain parallel statements nor can it invoke methods that contain parallel statements; the block execution must be sequential. In addition, each variable that is updated inside the atomic block must be in atomic access mode, when the atomic section is executed; access modes are explained in Section 11.

The following example shows below the use of atomic blocks for updating a doubly linked list:

```java
void node.delete() {
    Node: prev, next;
    atomic {
        prev = this.prev;
        next = this.next;
    }
}
```
if (prev != NULL) then prev.next = next;
if (next != NULL) then next.prev = prev;
}
}

Any access to the predecessor node or successor node and any update to this node done by another thread will occur either before the atomic block starts executing or after it has completed execution. The effect will be similar to the one achieved by using a synchronized statement; however, (a) one need not reference an explicit lock; the lock (or other synchronization mechanism) is implicit; and (b) PPL1 enforces the requirement that all accesses to variables that may conflict with accesses in the atomic block must be atomic (within atomic blocks) as well. As we shall see in Section 11, this requirement is enforced at compile time.

Discussion
Atomic blocks can be used to implement reduction operations: e.g., atomic \{a += myval\}, for a sum reduction. We should decide whether special syntax will be helpful here. Special syntax will indicate that all updates to a are sum reductions, allowing for a more efficient specialization; but, if we assume that the compiler can syntactically verify that all conflicting accesses are protected within an atomic block, that it will most frequently be able to detect that all accesses use the same operator, allowing for an optimization of the reduction.

Special syntax will still be useful for the special case where the global reduction is also a synchronization point – where it is best implemented as a collective operation; this case is handled by the reduction operators defined on arrays.

Missing Material
Need to decide on whether we want to support predefined and/or user-defined reduction operators and on the syntax and semantics for those.

11 Communication

11.1 Local and Global References

PPL1 variables come in two flavors: local variables can be accessed only at the site where they were created (allocated); while global variables can be accessed at any site. The type of a variable specifies whether it is local or global.

By default, a variable is local. This default can be overruled by specifying a suitable modifier in the type declaration for the variable.

The global modifier specifies that the variable can be accessed at any site.

The global modifier can also be used in the declaration of a class. If a class is global, then all its class and instance variables are global. Similarly, the global modifier can be used in the declaration of an array, in which case all the array components are global.
Consider the following example, adapted from the Java Language Specification document [15].

class Point {
    static int numPoints = 0;  // numPoints is a class variable
    int x, y;  // x,y are instance variables
    int[] w = new int[10];  // w[0] is array component
    Point() {
        numPoints++;
    }
    int setX(int x) {  // x is method parameter
        int oldx = this.x;  // oldx is local variable
        this.x = x;
        return oldx;
    }
}

Site[] s = new Site[3];
forall(int i : on s[i]) {
    Point p = new Point();  // exception is raised
    p.setX(2*i);
    ...
}

An instance of Point is created on each of the sites s[0], s[1] and s[2]. Each of these three instances has its own copy of the instance variables x, y and w; these can be accessed only on the site where the object was instantiated. This also holds true for the entries of the local instances of array w and for the argument and the local variable of method setX, when it is invoked.

By default, class variables are instantiated on the main site. Thus, the variable numPoints cannot be accessed at the sites s[0], s[1] and s[2]. When the Point constructor is executed on these sites then a run-time exception will be raised. This can be avoided by declaring the class variable to be global, as shown below.

class Point {
    global static int numPoints = 0;
    int x, y;
    int[] w = new int[10];
    Point() {
        atomic {numPoints++};
    }
    int setX(int x) {
        int oldx = this.x;
        this.x = x;
        return oldx;
    }
}

Site[] s = new Site[3];
forall(int i : on s[i]) {
global class Point {
    static int numPoints = 0;
    int x, y;
    int[] w = new int[10];
    Point() {
        atomic {numPoints++;}
    }
    int setX(int x) {
        int oldx = this.x;
        this.x = x;
        return oldx;
    }
}

global Point p = new Point();
Site[] s = new Site[3];
forall(int i : on s[i]) {
    int z = p.x + p.w[3] + numPoints;
    ...
}

Note that, in the example above, both Point and p are declared to be global. The global modifier in the declaration of Point indicates that the variables of an instance of Point can be accessed from any site; the global modifier in the declaration of p indicates that the variable p can be accessed from any site.

Consider the modified example below:

global class Point {
    static int numPoints = 0; // numPoints is a class variable
    int x, y; // x,y are instance variables
    int[] w = new int[10]; // w[0] is array component
    Point() {
        numPoints++;
    }
    int setX(int x) { // x is method parameter
        int oldx = this.x; // oldx is local variable
        this.x = x;
        return oldx;
    }
}
```java
}
}
global Point p = new Point();
Site[] s = new Site[3];
forall(int i : on s[i]) {
    Point myp = p;
    int z = myp.x+i;
    int[] myw = myp.w;
    ...
}
```

The variable `myp` is local, and can be accessed only at the site where it is instantiated; this variable is set at all three sites of the cohort `s` to contain a reference to the global `Point` object and can be used to access (global) instance variables of this object. Similarly, the variable `myw` is local, and can be accessed only at the site were it is instantiated; this variable is set at all three sites of the cohort `s` to contain a reference to the global array `w`.

Arrays can also be declared to be global, in which case all their components are global. Consider the example below:

```java
global int[][][] a = {{{1}},{{2,3}},{{4,5},{6}}}
Site[] s = new Site[s];
forall(int i : on s[i]) {
    int[][][] mya = a;
    int[][] myb = a[2];
    int[] myc = a[1,1];
    ...
}
```

Then, at each of the sites, `mya` will refer to the global array `a`; `myb` will refer to the global array `a[2]` (with two entries `myb[0] = {4,5} and myb[1] = {6}`); and `myc` will refer to the global array `a[1,1]` (with one entry `myc[0]=3`).

It must be the case that either all the components of a Java array are global, or all the components are local.

---

**Discussion**

We could relax the last restriction with more complex syntax; it is not obvious such relaxation is needed.

We could have more than two levels of locality; e.g., have variables that can be accessed only within a cohort; such a “cohort-local” variable would be accessible only within the cohort containing the site were it was instantiated. We postpone such an addition until it is shown to be needed.

Note that sites and cohorts are also local by default; in order to access a cohort at a site other than the site where it was instantiated it should be declared global. (An alternative design would have PPL1 support only local cohorts. This simplifies control, as the nesting of parallel constructs would follow simple scoping rules; but it also restricts expressiveness, e.g., the ability to collocate...
sites or to use sites for I/O.)

---

**Implementation Note**

The local/global modifier is part of a variable type; the Java type checking is extended to check, at compile time whenever possible, at run-time, otherwise, type mismatches.

On a distributed memory system, a PPL1 global reference will be a \([\text{site, local} \_\text{address}]\) pair. In a shared memory environment, the pair can be encoded in one memory address and variables are accessed via load/store operations; as shown later in this section, this will work even if the memory is not coherent, as PPL1 programs have well-defined synchronization points. In a distributed memory environment, global accesses are likely to be much more expensive than local accesses. Therefore, compiler optimizations that do bulk transfers and cache global data into local memory will be very important. The design of PPL1 facilitates such optimizations, as shown later in this section.

---

### 11.2 Access Modes

At any point in time, a variable is in one of the following access modes, with respect to an active site:

- **exclusive** If a variable is exclusive at a site than it can be accessed and updated by code executing at that site\(^{12}\).
- **shared** If a variable is shared at a site then it can be accessed, but not updated by code executing at the site. Any updating access will generate an exception.
- **transactional** If a variable is protected at a site then it can be accessed or updated within an atomic section by code executing at the site. An access that is not within an atomic section will generate an exception.
- **invalid** If a variable is invalid at a site than any access at the site to the variable will raise an exception.

Two accesses to a variable are said to conflict if one of them is updating the value of the variable. PPL1 prohibits conflicting accesses to a variable by two possibly concurrent statements (where possibly concurrent is as defined in Section 9.6).

The following three rules ensure that no conflicting accesses occur:

1. If a variable is exclusive at a site then it must be invalid at any other possibly concurrent site.
2. If a variable is shared at a site, then it must be either shared or invalid at any other possibly concurrent site.

\(^{12}\) Should we generate an exception if the variable is accessed within an atomic section?
3. If a variable is transactional at a site, then it must be either transactional or invalid at any other possibly concurrent site.

A PPL1 compiler must enforce these three rules.

When a global variable is declared, the declaration may include one or more of the access mode modifiers `exclusive`, `shared` or `transactional` that specify which access modes are supported for the variable. If no modifier is specified, then the default is that the variable is `exclusive` and can be accessed only at one site at a time.

We prioritize the four modes in the order `private` $\succ$ `shared` $\succ$ `protected` $\succ$ `invalid`.

The initial mode of a variable is determined by the mode modifiers associated with its declaration: The variable will be in the supported mode with the highest priority at the site where it was instantiated, and in `invalid` mode at all other sites. Thus, variables declared with no access mode modifiers are exclusive at the site where they are instantiated and invalid at other sites.

The mode of a variable can be explicitly changed by an access mode statement. The syntax is as follows:

```
AccessModeStatement:
   AccessMode SiteExpression : Variables ;
AccessMode:
   privat
   shared
   protected
Variables:
   Variable
   Variables , variable
```

`Variable` should be the of a variable that is valid in the scope where the statement occurs; `SiteExpression` should evaluate to a site reference or to a reference to a set of sites; frequently a `cohort` reference will be used as a `SiteExpression` (the usual conversion rules will convert the cohort map to the domain set).

The following example illustrates this syntax:

```java
global class Point {
   int x, y;
   Point(int x, int y) {this.x = x; this.y = y; }
   global static Point origin = new Point(0,0);
}
class Test {
   public static void main(String[] args) {
      global Point p = new Point(3,5);
      global Point q = p;
      Site s[] = new Site[3];
      shared s : origin); // origin can be concurrently accessed
      // on all sites of s
```
protected s: p.x, p.y ; // the coordinates of p can be accessed and
    // within an atomic section at all sites of s
private s[0]: q; // variable q can be accessed and updated
    // only on site s[0]
forall( int i: {..2}; on s[i]) {
    atomic{ p.x = i}; // the final value of p.x is either 0, 1 or 2
    atomic( p.y += i); // the final value of p.y is 8
    if (i==0) q = origin;
}

An access mode statement grants access in a given mode (exclusive, shared or transactional) to a variable at a site or a set of sites. By implication, it revokes access to this variable at all other sites, where the variable becomes invalid. The statement is erroneous if there is a possible access to the variable that is possibly concurrent with the execution of the access mode statement; informally, one cannot change the access mode of a variable while the variable is possibly accessed. Such errors are detected at compile time.

**Implementation Note**
The set of variables that are possibly accessed at each statement is determined by conservative compiler analysis. The “possibly-concurrent” relation is also determined by compiler analysis. The combination of the two is used to determine if a access mode statement is legal.

**Discussion**
We may want to support implicit access mode changing rules, where mode is changed according to program accesses: a variable becomes shared at a site if it is accessed (but not updated) at the site, and becomes exclusive if it is updated at the site; an exception is raised if access mode changes are inconsistent (e.g., a variable becomes exclusive at two concurrently executing sites). In other words, the user need not control data movement but need to ensure that there are no conflicting accesses.

The problem with such an approach is that it may encourage an inefficient programming style; more importantly, run-time detection of conflicting accesses can be expensive.

**Missing Material**
Need to provide a mechanism to change access mode for an object (i.e., for all variables of this object). Note that if \( p \) is a reference to an object, then \texttt{exclusive(s: p)} change the access mode for the variable \( p \), not for the object this variable refers to.

Need query methods to find what the status of a variable at a site is.

It is not obvious that the outlined approach for compile time detection of incorrect access mode changes can work efficiently, without being overly conservative. Will need to further study this issue and refine the design.
12 Partitioned Aggregates

The design presented in the last section allows a user to change the access mode of variables, one variable at a time. This is not convenient, in case one needs to change the access mode for entries of large aggregates, such as arrays. Furthermore, the resulting code may be inefficient: on many systems it is important to aggregate multiple small communications into large messages; a compiler may not be always capable of performing such an aggregation.

We provide in this section a mechanism to partition aggregates and to collectively modify the access mode of all partitions. This enables the implementation to move data more efficiently and to manage local caches more efficiently, on systems that do not have hardware support for caching.

The syntax for bulk changing of access mode to the components of an aggregate is similar to the syntax introduced in the previous section.

**BulkAccessModeStatement:**

```plaintext
BulkAccessModeStatement:
    AccessMode Cohort, Map : Maps ;
AccessMode:
    privat
    shared
    protected
Maps:
    Map
    Maps , Map
```

A statement of the form

```plaintext
mode c, m : a, b
```

will result in variables `a[p]` and `b[p]` being in access mode `mode` on each site `s=c[q]` with index `q` that belongs to the set `m[p]`, for each `p`. For the statement to be correct, then `a`, `b` and `m` should be conforming (have the same domain); and if the domain of cohort `c` is of type `X`, then the range of `m` is of type `SetofX`, and each value in a set `m[p]` must be a valid index of a site in the cohort `c`. In addition, if `mode = exclusive`, then the sets `m[p]` must be singletons, so that no element is mapped to two different sites.

This is illustrated in the following example:

```plaintext
... global shared int[][] a = new int({..[2,2]});  \ allocates 3x3 array of integers global exclusive float[] b = new float[{..[2,2]}];  \ allocates 3x3 array of floats Site[][] s = new Site({..[1,1]}); \ allocates 2x2 array of sites SetofPoint<2>[{..[2,2]}] u = {{..[1,1]}, {[0,0],[1,1]}, {}, {}, {}, {}, {}, {}, {}}, v = {[0,0],[0,1],[1,0],[1,1], {}, {}, {}, {}, {}, {}},
```
After the execution of this program fragment then $a[[0,0]]$ is shared by all four sites in cohort $s$; $a[[0,1]]$ is shared at sites $s[[0,0]]$ and $s[[1,1]]$; and no other entry of array $a$ can accessed on the sites of $s$. Similarly, $b[[i,j]]$ is exclusive on site $s[[i,j]]$, for $0 \leq i, j \leq 1$, while $b[[1,2]]$, $b[[2,1]]$ and $b[[2,2]]$ cannot be accessed on any site of cohort $s$.

Using this notation, one can change the distribution of components of an array ahead of the execution of a parallel construct, or during the execution, by using a barrier. Redistribution of array components may be expensive, if the distribution is computed on the fly. A user can make this redistribution more efficient, by providing as early as possible information on the map that is used to distribute the array (or map) elements:

- Whenever possible, one should use maps that do not change, i.e., maps that were declared final.
- Whenever possible, one should use maps that are defined using one of the map constructors described in the next section; it is expected that PPL1 compilers will be able to do a better job at analyzing these maps, as their general structure will be known at compile time.

### 12.1 Partition Map Constructors

**Block distribution:** Let $ss$ and $tt$ be two sets of type setof$S$ and setof$T$, respectively. Let $s_0, \ldots, s_m$ be the elements of $ss$ and let $t_0, \ldots, t_n$ be the elements of $t$, in the set order. Let $k = \lceil m/n \rceil$. Then the invocation

$$ss.$PartitionBlock(tt)$$

will return a mapping of type $\text{Map}<S, \text{setof}T>$ that maps each element $s_i$, where $k j \geq i < k(j + 1)$ into the singleton set $\{t_k\}$. In other words, the elements of $ss$ are divided into $n$ blocks of consecutive elements, all, with the possible exception of the last, of the same size; the elements of block $i$ are assigned to $t_i$.

One can also have block partitions with user defined block lengths. Let $\text{len}$ an array of int. Then the invocation

$$ss.$PartitionBlock(tt, len)$$

will return a mapping of type $\text{Map}<S, \text{setof}T>$ that maps each element $s_i$, where $\sum_{j\geq k} a[j] \geq i < \sum_{j\geq k+1} a[j]$ into the singleton set $\{t_k\}$. In other words, the elements of $ss$ are divided into $n$ blocks of consecutive elements, so that block $j$ has $a[j]$ elements; the elements of block $j$ are assigned to $t_j$. The size of array $\text{len}$ should equal the size of set $ss$.
**Block-cyclic Distribution:** With the same assumptions, as above, the invocation

\[ \text{ss.PartitionCyclic(tt,c)} \]

will return a mapping of type \( \text{Map<S, setofT}> \) that maps the elements \( s_0, \ldots, s_{c-1} \) to \( \{ t_0 \}; s_c, \ldots, s_{2c-1} \) to \( \{ t_1 \}; \) and, in general, if \( kc \leq j < (k+1)c \), then \( s_1 \) is mapped to \( \{ t_{k \mod n} \} \). If \( c=0 \) then and invocation to \( \text{ss.PartitionCyclic(tt,c)} \) returns the same result as an invocation to \( \text{s.PartitionBlock(t)} \).

**Distributions with ghost cells:** With the same assumptions as above, the invocation

\[ \text{ss.PartitionBlock(tt,g)} \]

where \( g \) is a positive integer, will return a mapping of type \( \text{Map<S, setofT}> \) that maps each element \( s_j \), into a set that contains all elements \( t_k \), where \( kj - g \geq i < k(j+1) + g \). In other words, \( t_k \) is associated with the same block as for a regular block distribution, but is also associated with \( g \) elements to the left of the block and with \( g \) elements to the right of the block. This distribution will map some elements of \( ss \) into subsets of \( tt \) that contain more than one element.

Block distributions with user defined block lengths, and block-cyclic distributions are extended in the same way.

**Grid distributions:** Let \( ss \) and \( tt \) be of type \( \text{Grid<N>} \). Let \( c \) be a vector of length \( N \). Assume that map returned by the invocation

\[ \{ \text{ss.min(i)}..\text{ss.max(i)} \}.\text{PartitionCyclic(} \{ \text{tt.min(i)}..\text{tt.max(i)} \},c[i]); \]

maps each \( k \), so that \( \text{ss.min(i)} \geq k \geq \text{ss.max(i)} \) into \( \{ f_1(k) \} \) (where \( \text{tt.min(i)} \leq f_i(k) \geq \text{tt.max(i)} \)). Then the invocation

\[ \text{ss.PartitionCyclic(tt,c)} \]

returns a map that maps point \( p = [p[1], \ldots, p[N]] \) into the singleton set \( \{ [f_1(p[1]), \ldots, f_N(p[N])] \} \). In other words, each dimension \( i \) is partitioned in a block cyclic distribution, as defined by parameter \( c[i] \); the resulting grid distribution is the direct product of these partitions in each dimension.

The definition can be extended to support ghost cells, by adding an extra array argument \( g \), so that \( g[i] \) specifies the overlap in dimension \( i \). Also, the definition can be extended to cover the case where \( ss \) is of type \( \text{Grid<N>} \) and \( tt \) is of type \( \text{Grid<M>} \), with \( M < N \); we use a special coordinate value, \text{maxint}, to indicate that the corresponding dimension is not to be partitioned, but projected, instead.

---

**Missing Material**

Need real examples.

May need to introduce additional partition constructors, as we see what’s needed for various examples.
Need to provide a way of specifying an initial distribution as an argument to a map or array constructor, so that the initial memory allocation for the array be distributed.

13 I/O

We assume that predefined sites are used to indicate I/O devices. To the least, the application environment will provide a one site cohort storage for persistent (disk) storage. Additional sites can be collocated with storage; each site that is collocated with storage corresponds to a file. The file name is provided as a site attribute, with key filename. We call such a site a file site.

I/O to this file is performed by copying data from or into an array or map that is associated with this site. At most one such an object can become associated with the site during program execution. Suppose that a file f becomes thus associated with a map a, and suppose that each entry of the map occupies w bytes of storage. Then the file is assumed to consist of successive w byte “words” that correspond to successive map entries, in index order.

A map becomes associated with a file if the map is instantiated on the site, or when an access mode statement is used to associate the map, or entries of the map, with the corresponding site. I.e., the map becomes associated with the file when a statement is executed that causes some entry of the map to be in a state other than invalid at the corresponding site. The state of the map entry at a file site can be invalid, or shared, but not exclusive or protected. I.e., a file site cannot write entries on a file, but can only hold them.

Assume that the map was instantiated at the file site. Then the map entries are initialized to contain the file values; the map constructor should allocate space for the map, but should not assign values to the entries.

Assume that the map was instantiated at another site. Then the initial value of the map entries is specified by the map constructor.

Whenever entries of the map become associated with the file site, then the current values of the entries are stored in the file. Note that an entry cannot be modified while it is associated with the file site; however, it can be dissociated from the file site and associated with another site in exclusive or protected model; the thread running and that site can then update the entry.

Discussion

One should think of the permanent file as of a cached value of the map or array that is maintained at the file site: the cached value may be initialized from the preexisting file; a file entry is updated whenever the communication protocol requires that the corresponding map entry be cached at the file site.

We may want to restrict threads that are associated with file sites so that be allowed to contain only object allocators, but no other executable statements.

The current design does not provide any strong typing for files: a file can be written as an array of characters and read as an array of floats. Additional type protection can be provided by associating suitable metadata with files – namely the information on the type of their entries.
Should support terminal I/O.

May want to support distributed files by associating a file not with a site, but with a cohort. This requires a mechanism for specifying an initial partition for an array. (Such a mechanism is desirable also for regular arrays that are not used for I/O.)

Need to provide examples.

References


